

# Modèle 3D d'un objet à partir de photographies: Extraction de silhouette

Martin Janin

13 juin 2017

Le travail effectué est celui prévu. La majorité de l'étude a été concentrée sur la détection de contours. La segmentation de l'image obtenue étant effectuée par le procédé simple présenté par Baumgart.

---

## Table des matières

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Choix d'implémentation</b>	<b>1</b>
2.1	Types et structures . . . . .	1
2.1.1	Type . . . . .	1
2.1.2	Structure . . . . .	2
2.1.3	Complexité . . . . .	2
<b>3</b>	<b>Procédé et Résultats</b>	<b>2</b>
<b>4</b>	<b>Conclusion</b>	<b>2</b>

---

## 1 Introduction

Le travail consiste en l'implémentation complète d'un script d'extraction de silhouette en python. Celle-ci a entraîné une réflexion importante sur l'optimisation du programme, de la manière d'implémenter les différentes étapes jusqu'au choix des types et structures utilisés. On détaillera d'abord les choix d'implémentation avant de présenter succinctement le procédé, trop complexe pour être complètement expliqué ici.

## 2 Choix d'implémentation

La totalité du programme est écrite en python. Il comprend certaines commandes en bash afin de suivre l'évolution de l'utilisation de la mémoire vive.

Les modules utilisés sont :

numpy	→ structure et opérations
system	→ suivie de la mémoire
matplotlib	→ tracé et affichage des images
itertools	→ outils sur les générateurs
time	→ suivi du temps d'exécution de chaque étape

### 2.1 Types et structures

Le programme prend en entrée une image RGB de taille de l'ordre du mégapixel. Traiter des données de cette taille motive des choix qui ne sont pas habituels dans d'autres domaines de l'informatique.

#### 2.1.1 Type

Le type retenu est le type float32 de numpy, conjugué naturellement au type complex64. C'est le meilleur choix pour plusieurs raisons :

- Les opérations sur les flottants sont plus rapides que sur les entiers. En effet, les processeurs sont d'abord conçus pour effectuer des opérations sur des nombres flottants.
- Les 8 bits alloués à l'exposant du type float32 permettent un nombre maximal de  $10^{256}$  ce qui évite tout dépassement de capacité.
- Il est bien plus aisé de normaliser une image de flottants puisque le type gère de lui même la perte de précision (contrairement aux entiers)
- Du fait des grandes complexités spatiales, il était impossible d'utiliser des flottants sur 64bits.

### 2.1.2 Structure

En ce qui concerne la structure adoptée, il s'agit des tableaux numpy, pour des raisons d'efficacité. On est amené à utiliser des tableaux à 5 dimensions, réparties comme suit :

- Les 3<sup>ème</sup> et 4<sup>ème</sup> correspondent au plan de l'image : indexer par rapport à ces dimensions correspond à choisir un pixel.
- Les deux premières dimensions correspondent au voisinage de chaque pixel. Chaque pixel contient en effet tout son voisinage afin de pouvoir effectuer des opérations dessus.
- La 5<sup>ème</sup> dimension est celle des composantes. On manipulera en effet des images composites, à l'instar d'une image RGB, ce qui permet d'effectuer les mêmes opérations sur toutes les composantes simultanément.

### 2.1.3 Complexité

Les choix d'implémentation ont principalement été motivés par l'optimisation de la complexité.

**Complexité temporelle.** En théorie, la meilleure manière d'effectuer des opérations sur plusieurs millions de pixels est de vectoriser les calculs, c'est-à-dire d'effectuer les calculs simultanément. On utilise pour cela des GPU (Graphical Process Unit) dont les circuits sont conçus pour effectuer simultanément des opérations sur des vecteurs. Cependant, programmer de manière efficace sur GPU nécessite une maîtrise de langages de bas niveau conçus spécialement pour. En pratique, les calculs vectoriels ont donc été faits avec numpy. Bien que numpy utilise uniquement le CPU pour toutes les opérations, quand les calculs sont correctement vectorisés, ils sont effectués par des boucles sous-jacentes codées en C. Les opérations vectorielles de numpy sont significativement plus rapides qu'en python et simule donc bien le comportement de calculs vectorisés.

**Complexité spatiale.** Vectoriser les calculs nécessite cependant de stocker simultanément les variables intermédiaires de millions de calculs dans la mémoire vive. Il en résulte une grande complexité spatiale. En effet, effectuer un calcul sur le voisinage de chaque pixel

demande :  $\text{TailleDesDonnees} \times \text{TailleDuVoisinage} \times \text{TailleDeLimage} \times \text{NombreDeComposantes}$  ce qui dans l'implémentation proposée est parfois de l'ordre de :  $80\text{Octet} \times 500\text{Mpix} \times 10$  soit 4Gio. Les ordinateurs actuels disposent d'une mémoire de l'ordre de 5Gio. L'optimisation spatiale est donc également importante. Comme il n'y a aucun moyen de gérer explicitement la mémoire en python, la solution mise en oeuvre a consisté à coder les fonctions clés de sorte que le calcul se face en place autant que possible.

**Coût de copie** Il faut noter enfin, que lorsqu'il s'agit d'effectuer des opérations sur le voisinage de chaque pixel, il est nécessaire de copier un grand nombre de fois l'image en mémoire (cf. structure). Le coût temporel de ces copies est non négligeable même avec l'optimisation sous-jacente de numpy. Cependant, au moyen d'une manipulation bas niveau des mémoires de travail vectorielles (registres vectoriels) d'un GPU, il serait possible de rendre ces coûts de copie négligeables.

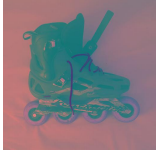






## 3 Procédé et Résultats

L'algorithme appliqué à l'image ci-dessous et les résultats obtenus sont présentés page 3



## 4 Conclusion

Le résultat obtenu, bien qu'imprécis permet ensuite, en combinant les silhouettes obtenues de plusieurs points de vues, de reconstituer le modèle 3D de l'objet. On a vu comment numpy pouvait permettre par la vectorisation (virtuelle), d'effectuer des opérations complexes sur des millions de données. L'optimisation spatiale et temporelle rigoureuse permet de mener à bien des calculs complexes (transformée de Fourier) en un temps raisonnable ( $\approx 45s$ )

	Opération	Description	Paramètres	Resultats
1	Conversion espace LAB	$(L = \frac{R+G+B}{2})$ code l'intensité, $(A = \frac{G-R+1}{2})$ et $(B = \frac{G-B+1}{2})$ la couleur. Cette espace est plus relevant pour détecter les bordures d'objet.		
2	Calcul des composantes de texture	On réalise la convolution du voisinage de chaque pixel par une gaussienne d'orientation et d'écart type variables. On calcul pour cela la transformée de Fourier de chaque voisinage	$\sigma = 4$ 4 directions	
3	Filtrage	On filtre toutes les composantes par des filtres de Canny (dérivée d'une gaussienne) d'orientation et d'écart type variables.	$\sigma \in \{1, 2\}$ , 8 directions	
4	Lissage parabolique	On approxime le voisinage de chaque pixel par une parabole $(ax^2 + bx + c)$ par la méthode des moindres carrés. On remplace l'image par l'image lissée : $I = \frac{c^+}{\text{distance au max local}} = c^+ \cdot \frac{2a^+}{ b }$	8 directions	
5	Sommation des réponses	On somme les 20 composantes obtenues $[(2\text{couleurs} + 4\text{orientation} * 2\text{couleurs}) * 2\text{écart type}]$ en les pondérant par des coefficients piochés dans la littérature.	coefficients de sommations	
6	Seuillage	On seuille l'image pour obtenir une image binaire	seuil = 0.2	
7	Opérations topologiques	On érode (tous les pixels non entourés sont supprimés) puis on dilate (tous les voisins des pixels sont allumés) successivement afin de fermer le contour de l'image	filtres de fermeture	
8	Approximation polygonale	On approxime le contour extérieur de l'objet par un polygone en séparant récursivement les segments d'un polygone jusqu'à atteindre un seuil d'approximation (distance maximale polygone-contour)	seuil d'approximation : 10 pixels	